BIAX Corporation v. Intel
Civil Action No. 2:05-cv-184-TJW

EXHIBIT 2
(PART 6)
FIRST AMENDED COMPLAINT FOR PATENT INFRINGEMENT

5,021,945

27

TABLE 12-continued

Main Program	Purpose
CALL SINE	Subroutine Call - Returns result in Reg R2
LOAD R2, R3	Temporarily save results in Reg R3
LOAD Y, R1	Load Y from memory into Reg R1 for parameter passing
CALL SINE	Subroutine Call - Returns result in Reg R2
MULT R2, R3, R4	Multiply Sin (x) with Sin (y) and store result in Reg R4
STORE R4, Z	Store final result in memory at Z

The SINE subroutine is set forth in Table 13:

TABLE 13

Instruction	(Subroutine	Purpose
I0	Load R1(L0), R2	Load Reg R2, level 1 with contents of Reg R1, level 0
Ip-1	(perform SINE), R7	Calculate SINE function and store result in Reg R7, level 1
Ip	Load R7, R2(L0)	Load Reg R2, level 0 with contents of Reg R7, level 1

Hence, under the teachings of the present invention and with reference to FIG. 14, instruction I0 of the subroutine loads register R2 of the current level (the subroutine's level or called level) with the contents of register R1 from the previous level (the calling routine or level). Note that the subroutine has a full set of registers with which to perform the processing independent of the register set of the calling routine. Upon completion of the subroutine call, instruction Ip causes register R7 of the current level to be stored in register R2 of the calling routine's level (which returns the results of the SINE routine back to the calling program's register set).

As described in more detail in connection with FIG. 22, the transfer between the levels occurs through the use of the SCSM dynamically generated information which can contain the absolute value of the current procedural level of the instruction (that is, the level of the called routine), the previous procedural level (that is, the level of the calling routine) and the context identifier. The absolute dynamic SCSM level information is generated by the LRD from the relative (static) SCSM information provided by the TOLL software. The context identifier is only used when processing a number of programs in a multi-user system. The relative SCSM information is shown in Table 13 for register R1 (of the calling routine) as R1(L0) and for register R2 as R2(L0). All registers of the current level have appended an implied (00) signifying the current procedural level.

This method and structure described in connection with FIGS. 13 and 14 differ substantially from prior art approaches where physical sharing of the same registers occurs between registers of a subroutine and its calling routine. By thereby limiting the number of registers that

28

are available for use by the subroutine, more system overhead for storing the registers in main memory is required. See, for example, the MIPS approach as set forth in "Reduced Instruction Set Computers" David A. Patterson, Communications of the ACM, January, 1985, Vol. 28, No. 1, Pgs. 8–21. In that reference, the first sixteen registers are local registers to be used solely by the subroutine, the next eight registers, registers 16 through 23, are shared between the calling routine and the subroutine, and final eight registers, registers 24 through 31 are shared between the global (or main) program and the subroutine. Clearly, out of 32 registers that are accessible by the subroutine, only 16 are dedicated solely for use by the subroutine in the processing of its program. In the processing of complex subroutines, the limited number of registers that are dedicated solely to the subroutine may not (in general) be sufficient for the processing of the subroutine. Data shuffling (entailing the storing of intermediate data in memory) must then occur, resulting in significant overhead in the processing of the routine.

Under the teachings of the present invention, the relative transfers between the levels which are known to occur at compile time are specified by adding the requisite information to the register identifiers as shown in FIG. 4 (the SCSM data), to appropriately map the instructions between the various levels. Hence, a completely independent set of registers is available to the calling routine and to each level of subroutine. The calling routine, in addition to accessing its own complete set of registers, can also gain direct access to a higher set of registers using the aforesaid static SCSM mapping code which is added to the instruction, as previously described. There is literally no reduction in the size of the register set available to a subroutine as specifically found in prior art approaches. Furthermore, the mapping code for the SCSM information can be a field of sufficient length to access any number of desired levels. For example, in one illustrated embodiment, a calling routine can access up to seven higher levels in addition to its own registers with a field of three bits. The present invention is not to be limited to any particular number of levels nor to any particular number of registers within a level. Under the teachings of the present invention, the mapping shown in FIG. 14 is a logical mapping and not a conventional physical mapping. For example, three levels, such as the calling routine level, the called level, and the global level require three bit maps. The relative identification of the levels can be specified by a two bit word in the static SCSM, for example, the calling routine by (00), the subordinate level by (01), and the global level by (11). Thus, each user's program is analyzed and the static SCSM relative procedural level information, also designated a window code, is added to the instructions prior to the issuance of the user program to a specific LRD. Once the user is assigned to a specific LRD, the static SCSM level information is used to generate the LRD dependent and dynamic SCSM information which is added as it is needed.

2. Detailed Description of the Hardware

As shown in FIG. 6, the TDA system 600 of the present invention is composed of memory 610, logical resource drivers (LRD) 620, processor elements (PEs) 640, and shared context storage files 660. The following detailed description starts with the logical resource

BIA0001188

5,021,945

29

drivers since the TOLL software output is loaded into this hardware.

a. Logical Resource Drivers (LRDs)

The details of a particular logical resource driver (LRD) is set forth in FIG. 15. As shown in FIG. 6, each logical resource driver 620 is interconnected to the LRD-memory network 630 on one side and to the processor elements 640 through the PE-LRD network 650 on the other side. If the present invention were a SIMD machine, then only one LRD is provided and only one context file is provided. For MIMD capabilities, one LRD and one context file is provided for each user so that, in the embodiment illustrated in FIG. 6, up to "n" users can be accommodated.

The logical resource driver 620 is composed of a data cache section 1500 and an instruction selection section 1510. In the instruction selection section, the following components are interconnected. An instruction cache address translation unit (ATU) 1512 is interconnected to the LRD-memory network 630 over a bus 1514. The instruction cache ATU 1512 is further interconnected over a bus 1516 to an instruction cache control circuit 1518. The instruction cache control circuit 1518 is interconnected over lines 1520 to a series of cache partitions 1522a, 1522b, 1522c, and 1522d. Each of the cache partitions is respectively connected over busses 1524a, 1524b, 1524c, and 1524d to the LRD-memory network 630. Each cache partition circuit is further interconnected over lines 1536a, 1536b, 1536c, and 1536d to a processor instruction queue (PIQ) bus interface unit 1544. The PIQ bus interface unit 1544 is connected over lines 1546 to a branch execution unit (BEU) 1548 which in turn is connected over lines 1550 to the PE-context file network 670. The PIQ bus interface unit 1544 is further connected over lines 1552a, 1552b, 1552c, and 1552d to a processor instruction queue (PIQ) buffer unit 1560 which in turn is connected over lines 1562a, 1562b, 1562c, and 1562d to a processor instruction queue (PIQ) processor assignment circuit 1570. The PIQ processor assignment circuit 1570 is in turn connected over lines 1572a, 1572b, 1572c, and 1572d to the PE-LRD network 650 and hence to the processor elements 640.

On the data cache portion 1500, a data cache ATU 1580 is interconnected over bus 1582 to the LRD-memory network 630 and is further interconnected over bus 1584 to a data cache control circuit 1586 and over lines 1588 to a data cache interconnection network 1590. The data cache control 1586 is also interconnected to data cache partition circuits 1592a, 1592b, 1592c and 1592d over lines 1593. The data cache partition circuits, in turn, are interconnected over lines 1594a, 1594b, 1594c, and 1594d to the LRD-memory network 630. Furthermore, the data cache partition circuits 1592 are interconnected over lines 1596a, 1596b, 1596c, and 1596d to the data cache interconnection network 1590. Finally, the data cache interconnection network 1590 is interconnected over lines 1598a, 1598b, 1598c, and 1598d to the PE-LRD network 650 and hence to the processor elements 640.

In operation, each logical resource driver (LRD) 620 has two sections, the data cache portion 1500 and the instruction selection portion 1510. The data cache portion 1500 acts as a high speed data buffer between the processor elements 640 and memory 610. Note that due to the number of memory requests that must be satisfied per unit time, the data cache 1500 is interleaved. All data requests made to memory by a processor element 640 are issued on the data cache interconnection net-

30

work 1590 and intercepted by the data cache 1592. The requests are routed to the appropriate data cache 1592 by the data cache interconnection network 1590 using the context identifier that is part of the dynamic SCSM information attached by the LRD to each instruction that is executed by the processors. The address of the desired datum determines in which cache partition the datum resides. If the requested datum is present (that is, a data cache hit occurs), the datum is sent back to the requesting processor element 640.

If the requested datum is not present in data cache, the address delivered to the cache 1592 is sent to the data cache ATU 1580 to be translated into a system address. The system address is then issued to memory. In response, a block of data from memory (a cache line or block) is delivered into the cache partition circuits 1592 under control of data cache control 1586. The requested data, that is resident in this cache block, is then sent through the data cache interconnection network 1590 to the requesting processor element 640. It is to be expressly understood that this is only one possible design. The data cache portion is of conventional design and many possible implementations are realizable to one skilled in the art. As the data cache is of standard functionality and design, it will not be discussed further.

The instruction selection portion 1510 of the LRD has three major functions; instruction caching, instruction queueing and branch execution. The system function of the instruction cache portion of selection portion 1510 is typical of any instruction caching mechanism. It acts as a high speed instruction buffer between the processors and memory. However, the current invention presents methods and an apparatus configuration for realizing this function that are unique.

One purpose of the instruction portion 1510 is to receive execution sets from memory, place the sets into the caches 1522 and furnish the instructions within the sets, on an as needed basis, to the processor elements 640. As the system contains multiple, generally independent, processor elements 640, requests to the instruction cache are for a group of concurrently executable instructions. Again, due to the number of requests that must be satisfied per unit time, the instruction cache is interleaved. The group size ranges from none to the number of processors available to the users. The groups are termed packets, although this does not necessarily imply that the instructions are stored in a contiguous manner. Instructions are fetched from the cache on the basis of their instruction firing time (IFT). The next instruction firing time register contains the firing time of the next packet of instructions to be fetched. This register may be loaded by the branch execution unit 1548 of the LRD as well as incremented by the cache control unit 1518 when an instruction fetch has been completed.

The next IFT register (NIFTR) is a storage register that is accessible from the context control unit 1518 and the branch execution unit 1548. Due to its simple functionality, it is not explicitly shown. Technically, it is a part of the instruction cache control unit 1518, and is further buried in the control unit 1660 (FIG. 16). The key point here is that the NIFTR is merely a storage register which can be incremented or loaded.

The instruction cache selection portion 1510 receives the instructions of an execution set from memory over bus 1524 and, in a round robin manner, places instructions word into each cache partitions, 1522a, 1522b, 1522c and 1522d. In other words, the instructions in the execution set are directed so that the first instruction is

BIA0001189

5,021,945

31

delivered to cache partition 1522a, the second instruction to cache partition 1522b, the third instruction to cache partition 1522c, and the fourth instruction to cache partition 1522d. The fifth instruction is then directed to cache partition 1522a, and so on until all of the instructions in the execution set are delivered into the cache partition circuits.

All the data delivered to the cache partitions are not necessarily stored in the cache. As will be discussed, the execution set header and trailer may not be stored. Each cache partition attaches a unique identifier (termed a tag) to all the information that is to be stored in that cache partition. The identifier is used to verify that information obtained from the cache is indeed the information desired.

When a packet of instructions is requested, each cache partition determines if the partition contains an instruction that is a member of the requested packet. If none of the partitions contain an instruction that is a member of the requested packet (that is, a miss occurs), the execution set that contains the requested packet is requested from memory in a manner analogous to a data cache miss.

If a hit occurs (that is, at least one of the partitions 1522 contains an instruction from the requested packet), the partition(s) attach any appropriate dynamic SCSM information to the instruction(s). The dynamic SCSM information, which can be attached to each instruction, is important for multi-user applications. The dynamically attached SCSM information identifies the context file (see FIG. 6) assigned to a given user. Hence, under the teachings of the present invention, the system 600 is capable of delay free switching among many user context files without requiring a master processor or access to memory.

The instruction(s) are then delivered to the PIQ bus interface unit 1544 of the LRD 620 where it is routed to the appropriate PIQ buffers 1560 according to the logical processor number (LPN) contained in the extended intelligence that the TOLL software, in the illustrated embodiment, has attached to the instruction. The instructions in the PIQ buffer unit 1560 are buffered for assignment to the actual processor elements 640. The processor assignment is performed by the PIQ processor assignment unit 1570. The assignment of the physical processor elements is performed on the basis of the number of processor elements currently available and the number of instructions that are available to be assigned. These numbers are dynamic. The selection process is set forth below.

The details of the instruction cache control 1518 and of each cache partition 1522 of FIG. 15 are set forth in FIG. 16. In each cache partition circuit 1522, five circuits are utilized. The first circuit is the header route circuit 1600 which routes an individual word in the header of the execution set over a path 1520b to the instruction cache context control unit 1660. The control of the header route circuit 1600 is effected over path 1520a by the header path select circuit 1602. The header path select circuit 1602 based upon the address received over lines 1520b from the control unit 1660 selectively activates the required number of header routers 1600 in the cache partitions. For example, if the execution set has two header words, only the first two header route circuits 1600 are activated by the header path select circuit 1602 and therefore two words of header information are delivered over bus 1520b to the control unit 1660 from the two activated header route circuits 1600

32

of cache partition circuits 1522a and 1522b (not shown). As mentioned, successive words in the execution set are delivered to successive cache partition circuits 1522.

For example, assume that the data of Table 1 represents an entire execution set and that appropriate header words appear at the beginning of the execution set. The instructions with the earliest instruction firing times (IFTs) are listed first and for a given IFT, those instructions with the lowest logical processor number are listed first. The table reads:

TABLE 14

Header Word 1		
Header Word 2		
I0	(T16)	(PE0)
I1	(T16)	(PE1)
I4	(T16)	(PE2)
I2	(T17)	(PE0)
I5	(T17)	(PE1)
I3	(T18)	(PE0)

Hence, the example of Table 1 (that is, the matrix multiply inner loop), now has associated with it two header words and the extended information defining the firing time (IFT) and the logical processor number (LPN). As shown in Table 14, the instructions were reordered by the TOLL software according to the firing times. Hence, as the execution set shown in Table 14 is delivered through the LRD-memory network 630 from memory, the first word (Header Word 1) is routed by partition CACHE0 to the control unit 1660. The second word (Header Word 2) is routed by partition CACHE1 (FIG. 15) to the control unit 1660. Instruction I0 is delivered to partition CACHE2, instruction I1 to partition CACHE3, instruction I2 to partition CACHE0, and so forth. As a result, the cache partitions 1522 now contain the instructions as shown in Table 15:

TABLE 15

Cache0	Cache1	Cache2	Cache3
		I0	I1
I4	I2	I5	I3

It is important to clarify that the above example has only one basic block in the execution set (that is, it is a simplistic example). In actuality, an execution set would have a number of basic blocks.

The instructions are then delivered for storage into a cache random access memory (RAM) 1610 resident in each cache partition. Each instruction is delivered from the header router 1600 over a bus 1602 to the tag attacher circuit 1604 and then over a line 1606 into the RAM 1610. The tag attacher circuit 1604 is under control of a tag generation circuit 1612 and is interconnected therewith over a line 1520c. Cache RAM 1610 could be a conventional cache high speed RAM as found in conventional superminicomputers.

The tag generation circuit 1612 provides a unique identification code (ID) for attachment to each instruction before storage of that instruction in the designated RAM 1610. The assigning of process identification tags to instructions stored in cache circuits is conventional and is done to prevent aliasing of the instructions. "Cache Memories" by Alan J. Smith, ACM Computing Surveys, Vol. 14, September, 1982. The tag comprises a sufficient amount of information to uniquely identify it from each other instruction and user. The illustrated

BIA0001190

33
34

instructions already include the IFT and LPN, so that subsequently, when instructions are retrieved for execution, they can be fetched based on their firing times. As shown in Table 16, below, each instruction containing the extended information and the hardware tag is stored, as shown, for the above example:

TABLE 16

CACHE0:	I4(T16) (PE2) (ID2)
CACHE1:	I2(T17) (PE0) (ID3)
CACHE2:	I0(T16) (PE0) (ID0)
	I5(T17) (PE1) (ID4)
CACHE3:	I1(T16) (PE1) (ID1)
	I3(T18) (PE0) (ID5)

As stated previously, the purpose of the cache partition circuits 1522 is to provide a high speed buffer between the slow main memory 610 and the fast processor elements 640. Typically, the cache RAM 1610 is a high speed memory capable of being quickly accessed. If the RAM 1610 were a true associative memory, as can be witnessed in Table 16, each RAM 1610 could be addressed based upon instruction firing times (IFTs). At the present time, such associative memories are not economically justifiable and an IFT to cache address translation circuit 1620 must be utilized. Such a circuit is conventional in design and controls the addressing of each RAM 1610 over a bus 1520d. The purpose of circuit 1620 is to generate the RAM address of the desired instructions given the instruction firing time. Hence, for instruction firing time T16, CACHE0, CACHE2, and CACHE3, as seen in Table 16, would produce instructions I4, I0, and I1 respectively.

When the cache RAMs 1610 are addressed, those instructions associated with a specific firing time are delivered over lines 1624 into a tag compare and privilege check circuit 1630. The purpose of the tag compare and privilege check circuit 1630 is to compare the hardware tags (ID) to generated tags to verify that the proper instruction has been delivered. The reference tag is generated through a second tag generation circuit 1632 which is interconnected to the tag compare and privilege check circuit 1630 over a line 1520e. A privilege check is also performed on the delivered instruction to verify that the operation requested by the instruction is permitted given the privilege status of the process (e.g., system program, application program, etc.). This is a conventional check performed by computer processors which support multiple levels of processing states. A hit/miss circuit 1640 determines which RAMs 1610 have delivered the proper instructions to the PIQ bus interface unit 1544 in response to a specific instruction fetch request.

For example, and with reference back to Table 16, if the RAMs 1610 are addressed by circuit 1620 for instruction firing time T16, CACHE0, CACHE2, and CACHE3 would respond with instructions thereby comprising a hit indication on those cache partitions. Cache 1 would not respond and that would constitute a miss indication and this would be determined by circuit 1640 over line 1520g. Thus, each instruction, for instruction firing time T16, is delivered over bus 1632 into the SCSM attacher 1650 wherein dynamic SCSM information, if any, is added to the instruction by an SCSM attacher 1650. For example, hardware 1650 can replace the static SCSM procedural level information (which is a relative value) with the actual procedural level values. The actual values are generataed

from a procedural level counter data and the static SCSM information.

When all of the instructions associated with an individual firing time have been read from the RAM 1610, the hit and miss circuit 1640 over lines 1646 informs the instruction cache control unit 1660 of this information. The instruction cache context control unit 1660 contains the next instruction firing time register, a part of the instruction cache control 1518 which increments the instruction firing time to the next value. Hence, in the example, upon the completion of reading all instructions associated with instruction firing time T16, the instruction cache context control unit 1660 increments to the next firing time, T17, and delivers this information over lines 1664 to an access resolution circuit 1670, and over lines 1520f to the tag compare and privilege check circuit 1630. Also note that there may be firing times which have no valid instructions, possibly due to operational dependencies detected by the TOLL software. In this case, no instructions would be fetched from the cache and transmitted to the PIQ interface.

The access resolution circuit 1670 coordinates which circuitry has access to the instruction cache RAMs 1610. Typically, these RAMs can satisfy only a single request at each clock cycle. Since there could be two requests to the RAMs at one time, an arbitration method must be implemented to determine which circuitry obtains access. This is a conventional issue in the design of cache memory, and the access resolution circuit resolves the priority question as is well known in the field.

The present invention can and preferably does support several users simultaneously in both time and space. In previous prior art approaches (CDC, IBM, etc.), multi-user support was accomplished solely by timesharing the processor(s). In other words, the processors were shared in time. In this system, multi-user support is accomplished (in space) by assigning an LRD to each user that is given time on the processor elements. Thus, there is a spatial aspect to the sharing of the processor elements. The operating system of the machine deals with those users assigned to the same LRD in a timeshared manner, thereby adding the temporal dimension to the sharing of the processors.

Multi-user support is accomplished by the multiple LRDs, the use of plural processor elements, and the multiple context files 660 supporting the register files and condition code storage. As several users may be executing in the processor elements at the same time, additional pieces of information must be attached to each instruction prior to its execution to uniquely identify the instruction source and any resources that it may use. For example, a register identifier must contain the absolute value of the subroutine procedural level and the context identifier as well as the actual register number. Memory addresses must also contain the LRD identifier from which the instruction was issued to be properly routed through the LRD-Memory interconnection network to the appropriate data cache.

The additional and required information comprises two components, a static and a dynamic component; and the information is termed "shared context storage mapping" (SCSM). The static information results from the compiler output and the TOLL software gleans the information from the compiler generated instruction stream and attaches the register information to the instruction prior to its being received by an LRD.

BIA0001191

35   36

The dynamic information is hardware attached to the instruction by the LRD prior to its issuance to the processors. This information is composed of the context/LRD identifier corresponding to the LRD issuing the instruction, the absolute value of the current procedural level of the instruction, the process identifier of the current instruction stream, and preferably the instruction status information that would normally be contained in the processors of a system having processors that are not context free. This later information would be composed of error masks, floating point format modes, rounding modes, and so on.

In the operation of the circuitry in FIG. 16, one or more execution sets are delivered into the instruction cache circuitry. The header information for each set is delivered to one or more successive cache partitions and is routed to the context control unit 1660. The instructions in the execution set are then individually, on a round robin basis, routed to each successive cache partition unit 1522. A hardware identification tag is attached to each instruction and the instruction is then stored in RAM 1610. As previously discussed, each execution set is of sufficient length to minimize instruction cache defaults and the RAM 1610 is of sufficient size to store the execution sets. When the processor elements require the instructions, the number and cache locations of the valid instructions matching the appropriate IFTs are determined. The instructions stored in the RAM's 1610 are read out; the identification tags are verified; and the privilege status checked. The instructions are then delivered to the PIQ bus interface unit 1544. Each instruction that is delivered to the PIQ bus interface unit 1544, as is set forth in Table 17, includes the identification tag (ID) and the hardware added SCSM information.

TABLE 17

CACHE0:	I4(T16) (PE2) (ID2) (SCSM0)
CACHE1:	I2(T17) (PE0) (ID3) (SCSM1)
CACHE2:	I0(T16) (PE0) (ID0) (SCSM2)
	I5(T17) (PE1) (ID4) (SCSM3)
CACHE3:	I1(T16) (PE1) (ID1) (SCSM4)
	I3(T18) (PE0) (ID5) (SCSM5)

If an instruction is not stored in RAM 1610, a cache miss occurs and a new execution set containing the instruction is read from main memory over lines 1523.

In FIG. 17, the details of the PIQ bus interface unit 1544 and the PIQ buffer unit 1560 are set forth. Referring to FIG. 17, the PIQ bus interface unit 1544 receives instructions as set forth in Table 17, above, over lines 1536. A search tag hardware 1702 has access to the value of the present instruction firing time over lines 1549 and searches the cache memories 1522 to determine the address(es) of those registers containing instructions having the correct firing times. The search tag hardware 1702 then makes available to the instruction cache control circuitry 1518 the addresses of those memory locations for determination by the instruction cache control of which instructions to next select for delivery to the PIQ bus interface 1544.

These instructions access, in parallel, a two-dimensional array of bus interface units (BIU's) 1700. The bus interface units 1700 are interconnected in a full access non-blocking network by means of connections 1710 and 1720, and connect over lines 1552 to the PIQ buffer unit 1560. Each bus interface unit (BIU) 1700 is a conventional address comparison circuit composed of: TI

74L85 4 bit magnitude comparators, Texas Instruments Company, P.O. Box 225012, Dallas, Tex. 75265. In the matrix multiply example, for instruction firing time T16, CACHE0 contains instruction I4 and CACHE3 (corresponding to CACHE n in FIG. 17) contains instruction I1. The logical processor number assigned to instruction I4 is PE2. The logical processor number PE2 activates a select (SEL) signal of the bus interface unit 1700 for processor instruction queue 2 (this is the BIU3 corresponding to the CACHE0 unit containing the instruction). In this example, only that BIU3 is activated and the remaining bus interface units 1700 for that BIU3 row and column are not activated. Likewise, for CACHE3 (CACHE n in FIG. 17), the corresponding BIU2 is activated for processor instruction QUEUE 1.

The PIQ buffer unit 1560 is comprised of a number of processor instruction queues 1730 which store the instructions received from the PIQ bus interface unit 1544 in a first in-first out (FIFO) fashion as shown in Table 18:

TABLE 18

PIQ0	PIQ1	PIQ2	PIQ3
I0	I1	I4	—
I2	—	—	—
I3	—	—	—

In addition to performing instruction queueing functions, the PIQ's 1730 also keep track of the execution status of each instruction that is issued to the processor elements 640. In an ideal system, instructions could be issued to the processor elements every clock cycle without worrying about whether or not the instructions have finished execution. However, the processor elements 640 in the system may not be able to complete an instruction every clock cycle due to the occurrence of exceptional conditions, such as a data cache miss and so on. As a result, each PIQ 1730 tracks all instructions that it has issued to the processor elements 640 that are still in execution. The primary result of this tracking is that the PIQ's 1730 perform the instruction clocking function for the LRD 620. In other words, the PIQ's 1730 determine when the next firing time register can be updated when executing straightline code. This in turn begins a new instruction fetch cycle.

Instruction clocking is accomplished by having each PIQ 1730 form an instruction done signal that specifies that the instruction(s) issued by a given PIQ either have executed or, in the case of pipelined PE's, have proceeded to the next stage. This is then combined with all other PIQ instruction done signals from this LRD and is used to gate the increment signal that increments the next firing time register. The "done" signals are delivered over lines 1564 to the instruction cache control 1518.

Referring to FIG. 18, the PIQ processor assignment circuit 1570 contains a two dimensional array of network interface units (NIU's) 1800 interconnected as a full access switch to the PE-LRD network 650 and then to the various processor elements 640. Each network interface unit (NIU) 1800 is comprised of the same circuitry as the bus interface units (BIU) 1700 of FIG. 17. In normal operation, the processor instruction queue #0 (PIQ0) can directly access processor element 0 by activating the NIU0 associated with the column corresponding to queue #0, the remaining network interface units NIU0, NIU1, NIU2, NIU3 of the PIQ processor

BIA0001192

5,021,945

37

38

alignment circuit for that column and row being deactivated. Likewise, processor instruction queue #3 (PIQ3) normally accesses processor element 3 by activating the NIU3 of the column associated with queue #3, the remaining NIU0, NIU1, NIU2, and NIU3 of that column and row being deactivated. The activation of the network interface units 1800 is under the control of an instruction select and assignment unit 1810.

Unit 1810 receives signals from the PIQ's 1730 within the LRD that the unit 1810 is a member of over lines 1811, from all other units 1810 (of other LRD's) over lines 1813, and from the processor elements 640 through the network 650. Each PIQ 1730 furnishes the unit 1810 with a signal that corresponds to "I have an instruction that is ready to be assigned to a processor." The other PIQ buffer units furnish this unit 1810 and every other unit 1810 with a signal that corresponds to "My PIQ 1730 (#x) has an instruction ready to be assigned to a processor." Finally, the processor elements furnish each unit 1810 in the system with a signal that corresponds to "I can accept a new instruction."

The unit 1810 on an LRD transmits signals to the PIQs 1730 of its LRD over lines 1811, to the network interface units 1800 of its LRD over lines 1860 and to the other units 1810 of the other LRDs in the system over lines 1813. The unit 1810 transmits a signal to each PIQ 1730 that corresponds to "Gate your instruction onto the PE-LRD interface bus (650)." The unit transmits a select signal to the network interface units 1800. Finally, the unit 1810 transmits a signal that corresponds to "I have used processor element #x" to each other unit 1810 in the system for each processor which it is using.

In addition, each unit 1810 in each LRD has associated with it a priority that corresponds to the priority of the LRD. This is used to order the LRDs into an ascending order from zero to the number of LRDs in the system. The method used for assigning the processor elements is as follows. Given that the LRDs are ordered, many allocation schemes are possible (e.g., round robin, first come first served, time slice, etc.). However, these are implementation details and do not impact the functionality of this unit under the teachings of the present invention.

Consider the LRD with the current highest priority. This LRD gets all the processor elements that it requires and assigns the instructions that are ready to be executed to the available processor elements. If the processor elements are context free, the processor elements can be assigned in any manner whatsoever. Typically, however, assuming that all processors are functioning correctly, instructions from PIQ #0 are routed to processor element #0, provided of course, processor element #0 is available.

The unit 1810 in the highest priority LRD then transmits this information to all other units 1810 in the system. Any processors left open are then utilized by the next highest priority LRD with instructions that can be executed. This allocation continues until all processors have been assigned. Hence, processors may be assigned on a priority basis in a daisy chained manner.

If a particular processor element, for example, element 1 has failed, the instruction selective assignment unit 1810 can deactivate that processor element by deactivating all network instruction units NIU1. It can then, through hardware, reorder the processor elements so that, for example, processor element 2 receives all instructions logically assigned to processor element 1,

processor element 3 is now assigned to receive all instructions logically assigned to processor 2, etc. Indeed, redundant processor elements and network interface units can be provided to the system to provide for a high degree of fault tolerance.

Clearly, this is but one possible implementation. Other methods are also realizable.

b. Branch Execution Unit (BEU)

Referring to FIG. 19, the Branch Execution Unit (BEU) 1548 is the unit in the present invention responsible for the execution of all branch instructions which occur at the end of each basic block. There is, in the illustrated embodiment, one BEU 1548 for each supported context and so, with reference to FIG. 6, "n" supported contexts require "n" BEU's. The illustrated embodiment uses one BEU for each supported context because each BEU 1548 is of simple design and, therefore, the cost of sharing a BEU between plural contexts would be more expensive than allowing each context to have its own BEU.

The BEU 1548 executes branches in a conventional manner with the exception that the branch instructions are executed outside the PE's 640. The BEU 1548 evaluates the branch condition and, when the target address is selected, generates and places this address directly into the next instruction fetch register. The target address generation is conventional for unconditional and conditional branches that are not procedure calls or returns. The target address can be (a) taken directly from the instruction, (b) an offset from the current contents of the next instruction fetch register, or (c) an offset of a general purpose register of the context register file.

A return branch from a subroutine is handled in a slightly different fashion. To understand the subroutine return branch, discussion of the subroutine call branch is required. When the branch is executed, a return address is created and stored. The return address is normally the address of the instruction following the subroutine call. The return address can be stored in a stack in memory or in other storage local to the branch execution unit. In addition, the execution of the subroutine call increments the procedural level counter.

The return from a subroutine branch is also an unconditional branch. However, rather than containing the target address within the instruction, this type of branch reads the previously stored return address from storage, decrements the procedural level counter, and loads the next instruction fetch register with the return address. The remainder of the disclosure discusses the evaluation and execution of conditional branches. It should be noted that the techniques described also apply to unconditional branches, since these are, in effect, conditional branches in which the condition is always satisfied. Further, these same techniques also apply to the subroutine call and return branches, which perform the additional functions described above.

To speed up conditional branches, the determination of whether a conditional branch is taken or not, depends solely on the analysis of the appropriate set of condition codes. Under the teachings of the present invention, no evaluation of data is performed other than to manipulate the condition codes appropriately. In addition, an instruction, which generates a condition code that a branch will use, can transmit the code to BEU 1548 as well as to the condition code storage. This eliminates the conventional extra waiting time required for the

BIA0001193

5,021,945

39

code to become valid in the condition code storage prior to a BEU being able to fetch it.

The present invention also makes extensive use of delayed branching to guarantee program correctness. When a branch has executed and its effects are being propagated in the system, all instructions that are within the procedural domain of the branch must either have been executed or be in the process of being executed, as discussed in connection with the example of Table 6. In other words, changing the next-instruction pointer (in response to the branch) takes place after the current firing time has been updated to point to the firing time that follows the last (temporally executed) instruction of the branch. Hence, in the example of Table 6, instruction I5 at firing time T17 is delayed until the completion of T18 which is the last firing time for this basic block. The instruction time for the next basic block is then T19.

The functionality of the BEU **1548** can be described as a four-state state machine:

Stage 1:	Instruction decode
	- Operation decode
	- Delay field decode
	- Condition code access decode
Stage 2:	Condition code fetch/receive
Stage 3:	Branch operation evaluation
Stage 4:	Next instruction fetch location and firing time update

Along with determining the operation to be performed, the first stage also determines how long fetching can continue to take place after receipt of the branch by the BEU, and how the BEU is to access the condition codes for a conditional branch, that is, are they received or fetched.

Referring to FIG. 19, the branch instruction is delivered over bus **1546** from the PIQ bus interface unit **1544** into the instruction register **1900** of the BEU **1548**. The fields of the instruction register **1900** are designated as: FETCH/ENABLE, CONDITION CODE ADDRESS, OP CODE, DELAY FIELD, and TARGET ADDRESS. The instruction register **1900** is connected over lines **1910a** and **1910b** to a condition code access unit **1920**, over lines **1910c** to an evaluation unit **1930**, over lines **1910d** to a delay unit **1940**, and over lines **1910e** to a next instruction interface **1950**.

Once an instruction has been issued to BEU **1548** from the PIQ bus interface **1544**, instruction fetching must be held up until the value in the delay field has been determined. This value is measured relative to the receipt of the branch by the BEU, that is stage 1. If there are no instructions that may be overlapped with this branch, this field value is zero. In this case, instruction fetching is held up until the outcome of the branch has been determined. If this field is non-zero, instruction fetching may continue for a number of firing times given by the value in this field.

The condition code access unit **1920** is connected to the register file - PE network **670** over lines **1550** and to the evaluation unit **1930** over lines **1922**. During stage 2 operation, the condition code access decode unit **1920** determines whether or not the condition codes must be fetched by the instruction, or whether the instruction that determines the branch condition delivers them. As there is only one instruction per basic block that will determine the conditional branch, there will never be more than one condition code received by the BEU for

40

a basic block. As a result, the actual timing of when the condition code is received is not important. If it comes earlier than the branch, no other codes will be received prior to the execution of the branch. If it comes later, the branch will be waiting and the codes received will always be the right ones. Note that the condition code for the basic block can include plural codes received at the same or different times by the BEU.

The evaluation unit **1930** is connected to the next instruction interface **1950** over lines **1932**. The next instruction interface **1950** is connected to the instruction cache control circuit **1518** over lines **1549** and to the delay unit **1940** over lines **1942**; and the delay unit **1940** is also connected to the instruction cache control unit **1518** over lines **1549**.

During the evaluation stage of operation, the condition codes are combined according to a Boolean function that represents the condition being tested. In the final stage of operation, either fetching of the sequential instruction stream continues, if a conditional branch is not taken, or the next instruction pointer is loaded, if the branch is taken.

The impact of a branch in the instruction stream can be described as follows. Instructions, as discussed, are sent to their respective PIQ's **1730** by analysis of the resident logical processor number (LPN). Instruction fetching can be continued until a branch is encountered, that is, until an instruction is delivered to the instruction register **1900** of the BEU **1548**. At this point, in a conventional system without delayed branching, fetching would be stopped until the resolution of the branch instruction is complete. See, for example, "Branch Prediction Strategies and Branch Target Buffer Design", J. F. K. Lee & A. J. Smith, IEEE Computer Magazine, January, 1984.

In the present system, which includes delayed branching, instructions must continue to be fetched until the next instruction fetched is the last instruction of the basic block to be executed. The time that the branch is executed is then the last time that fetching of an instruction can take place without a possibility of modifying the next instruction address. Thus, the difference between when the branch is fetched and when the effects of the branch are actually felt corresponds to the number of additional firing time cycles during which fetching can be continued.

The impact of this delay is that the BEU **1548** must have access to the next instruction firing time register of the cache controller **1518**. Further, the BEU **1548** can control the initiation or disabling of the instruction fetch process performed by the instruction cache control unit **1518**. These tasks are accomplished by signals over bus **1549**.

In operation the branch execution unit (BEU) **1548** functions as follows. The branch instruction, such as instruction I5 in the example above, is loaded into the instruction register **1900** from the PIQ bus interface unit **1544**. The contents of the instruction register then control the further operation of BEU **1548**. The FETCH-ENABLE field indicates whether or not the condition code access unit **1920** should retrieve the condition code located at the address stored in the CC-ADX field (called FETCH) or whether the condition code will be delivered by the generating instruction.

If a FETCH is requested, the unit **1920** accesses the register file-PE network **670** (see FIG. 6) to access the condition code storage **2000** which is shown in FIG. 20.

BIA0001194

41

Referring to FIG. 20, the condition code storage 2000, for each context file, is shown in the generalized case. A set of registers CC_{xy} are provided for storing condition codes for procedural level y. Hence, the condition code storage 2000 is accessed and addressed by the unit 1920 to retrieve, pursuant to a FETCH request, the necessary condition code. The actual condition code and an indication that the condition code is received by the unit 1920 is delivered over lines 1922 to the evaluation unit 1930. The OPCODE field, delivered to the evaluation unit 1930, in conjunction with the received condition code, functions to deliver a branch taken signal over line 1932 to the next instruction interface 1950. The evaluation unit 1930 is comprised of standard gate arrays such as those from LSI Logic Corporation, 1551 McCarthy Blvd., Milpitas, Calif. 95035.

The evaluation unit 1930 accepts the condition code set that determines whether or not the conditional branch is taken, and under control of the OPCODE field, combines the set in a Boolean function to generate the conditional branch taken signal.

The next instruction interface 1950 receives the branch target address from the TARGET-ADX field of the instruction register 1900 and the branch taken signal over line 1932. However, the interface 1950 cannot operate until an enable signal is received from the delay unit 1940 over lines 1942.

The delay unit 1940 determines the amount of time that instruction fetching can be continued after the receipt of a branch instruction by the BEU. Previously, it has been described that when a branch instruction is received by the BEU, instruction fetching continues for one more cycle and then stops. The instruction fetched during this cycle is held up from passing through PIQ bus interface unit 1544 until the length of the delay field has been determined. For example, if the delay field is zero (implying that the branch is to be executed immediately), these instructions must still be withheld from the PIQ bus buffer unit until it is determined whether or not these are the right instructions to be fetched. If the delay field is non-zero, the instructions would be gated into the PIQ buffer unit as soon as the delay value was determined to be non-zero. The length of the delay is obtained from DELAY field of the instruction register 1900. The delay unit receives the delay length from register 1900 and clock impulses from the context control 1518 over lines 1549. The delay unit 1940 decrements the value of the delay at each clock pulse; and when fully decremented, the interface unit 1950 becomes enabled.

Hence, in the discussion of Table 6, instruction I5 is assigned a firing time T17 but is delayed until firing time T18. During the delay time, the interface 1950 signals the instruction cache control 1518 over line 1549 to continue to fetch instructions to finish the current basic block. When enabled, the interface unit 1950 delivers the next address (that is, the branch execution address) for the next basic block into the instruction cache control 1518 over lines 1549.

In summary and for the example on Table 6, the branch instruction I5 is loaded into the instruction register 1900 during time T17. However, a delay of one firing time (DELAY) is also loaded into the instruction register 1900 as the branch instruction cannot be executed until the last instruction I3 is processed during time T18. Hence, even though the instruction I5 is loaded in register 1900, the branch address for the next basic block, which is contained in the TARGET AD-

42

DRESS, does not become effective until the completion of time T18. In the meantime, the next instruction interface 1950 issues instructions to the cache control 1518 to continue processing the stream of instructions in the basic block. Upon the expiration of the delay, the interface 1950 is enabled, and the branch is executed by delivering the address of the next basic block to the instruction cache control 1518.

Note that the delay field is used to guarantee the execution of all instructions in the basic block governed by this branch in single cycle context free PE's. A small complexity is encountered when the PE's are pipelined. In this case, there exist data dependencies between the instructions from the basic block just executed, and the instructions from the basic block to be executed. The TOLL software can analyze these dependencies when the next basic block is only targeted by the branch from this basic block. If the next basic block is targeted by more than one branch, the TOLL software cannot resolve the various branch possibilities and lets the pipelines drain, so that no data dependencies are violated. One mechanism for allowing the pipelines to drain is to insert NO-OP (no operation) instructions into the instruction stream. An alternate method provides an extra field in the branch instruction which inhibits the delivery of new instructions to the processor elements for a time determined by the data in the extra field.

c. Processor Elements (PE)

So far in the discussions pertaining to the matrix multiply example, a single cycle processor element has been assumed. In other words, an instruction is issued to the processor element and the processor element completely executes the instruction before proceeding to the next instruction. However, greater performance can be obtained by employing pipelined processor elements. Accordingly, the tasks performed by the TOLL software change slightly. In particular, the assignment of the processor elements is more complex than is shown in the previous example; and the hazards that characterize a pipeline processor must be handled by the TOLL software. The hazards that are present in any pipelined processor manifest themselves as a more sophisticated set of data dependencies. This can be encoded into the TOLL software by one practiced in the art. See for example, T. K. R. Gross, Stanford University, 1983, "Code Optimization of Pipeline Constraints", Doctorate Dissertation Thesis.

The assignment of the processors is dependent on the implementation of the pipelines and again, can be performed by one practiced in the art. A key parameter is determining how data is exchanged between the pipelines. For example, assume that each pipeline contains feedback paths between its stages. In addition, assume that the pipelines can exchange results only through the register sets 660. Instructions would be assigned to the pipelines by determining sets of dependent instructions that are contained in the instruction stream and then assigning each specific set to a specific pipeline. This minimizes the amount of communication that must take place between the pipelines (via the register set), and hence speeds up the execution time of the program. The use of the logical processor number guarantees that the instructions will execute on the same pipeline.

Alternatively, if there are paths available to exchange data between the pipelines, dependent instructions may be distributed across several pipeline processors instead of being assigned to a single pipeline. Again, the use of multiple pipelines and the interconnection network

5,021,945

43

between them that allows the sharing of intermediate results manifests itself as a more sophisticated set of data dependencies imposed on the instruction stream. Clearly, the extension of the teachings of this invention to a pipelined system is within the skill of one practiced in the art.

Importantly, the additional data (chaining) paths do not change the fundamental context free nature of the processor elements of the present invention. That is, at any given time (for example, the completion of any given instruction cycle), the entire process state associated with a given program (that is, context) is captured completely external to the processor elements. Data chaining results merely in a transitory replication of some of the data generated within the processor elements during a specific instruction clock cycle.

Referring to FIG. 21, a particular processor element 640 has a four-stage pipeline processor element. All processor elements 640 according to the illustrated embodiment are identical. It is to be expressly understood, that any prior art type of processor element such as a micro-processor or other pipeline architecture could not be used under the teachings of the present invention, because such processors retain substantial state information of the program they are processing. However, such a processor could be programmed with software to emulate or simulate the type of processor necessary for the present invention.

The design of the processor element is determined by the instruction set architecture generated by the TOLL software and, therefore, from a conceptual viewpoint, is the most implementation dependent portion of this invention. In the illustrated embodiment shown in FIG. 21, each processor element pipeline operates autonomously of the other processor elements in the system. Each processor element is homogeneous and is capable, by itself, of executing all computational and data memory accessing instructions. In making computational executions, transfers are from register to register and for memory interface instructions, the transfers are from memory to registers or from registers to memory.

Referring to FIG. 21, the four-stage pipeline for the processor element 640 of the illustrated embodiment includes four discrete instruction registers 2100, 2110, 2120, and 2130. Each processor element also includes four stages: stage 1, 2140; stage 2, 2150; stage 3, 2160, and stage 4, 2170. The first instruction register 2100 is connected through the network 650 to the PIQ processor assignment circuit 1570 and receives that information over bus 2102. The instruction register 2100 then controls the operation of stage 1 which includes the hardware functions of instruction decode and register 0 fetch and register 1 fetch. The first stage 2140 is interconnected to the instruction register over lines 2104 and to the second instruction register 2110 over lines 2142. The first stage 2140 is also connected over a bus 2144 to the second stage 2150. Register 0 fetch and register 1 fetch of stage 1 are connected over lines 2146 and 2148, respectively, to network 670 for access to the register file 660.

The second instruction register 2110 is further interconnected to the third instruction register 2120 over lines 2112 and to the second stage 2150 over lines 2114. The second stage 2150 is also connected over a bus 2152 to the third stage 2160 and further has the memory write (MEM WRITE) register fetch hardware interconnected over lines 2154 to network 670 for access to the register file 660 and its condition code (CC) hard-

44

ware connected over lines 2156 through network 670 to the condition code storage of context file 660.

The third instruction register 2120 is interconnected over lines 2122 to the fourth instruction register 2130 and is also connected over lines 2124 to the third stage 2160. The third stage 2160 is connected over a bus 2162 to the fourth stage 2170 and is further interconnected over lines 2164 through network 650 to the data cache interconnection network 1590.
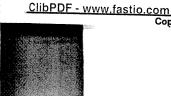
Finally, the fourth instruction register 2130 is interconnected over lines 2132 to the fourth stage, and the fourth stage has its store hardware (STORE) output connected over lines 2172 and its effective address update (EFF. ADD.) hardware circuit connected over lines 2174 to network 670 for access to the register file 660. In addition, the fourth stage has its condition code store (CC STORE) hardware connected over lines 2176 through network 670 to the condition code storage of context file 660.

The operation of the four-stage pipeline shown in FIG. 21 will now be discussed with respect to the example of Table 1 and the information contained in Table 19 which describes the operation of the processor element for each instruction.

TABLE 19

	Instruction IO, (I1):
Stage 1	Fetch Reg to form Mem-adx
Stage 2	Form Mem-adx
Stage 3	Perform Memory Read
Stage 4	Store RO, (R1)
	Instruction I2:
Stage 1	Fetch Reg RO and R1
Stage 2	No-Op
Stage 3	Perform multiply
Stage 4	Store R2 and CC
	Instruction I3:
Stage 1	Fetch Reg R2 and R3
Stage 2	No-Op
Stage 3	Perform addition
Stage 4	Store R3 and CC
	Instruction I4:
Stage 1	Fetch Reg R4
Stage 2	No-Op
Stage 3	Perform decrement
Stage 4	Store R4 and CC

For instructions I0 and I1, the performance by the processor element 640 in FIG. 21 is the same except in stage 4. The first stage is to fetch the memory address from the register which contains the address in the register file. Hence, stage 1 interconnects circuitry 2140 over lines 2146 through network 670 to that register and downloads it into register 0 from the interface of stage 1. Next, the address is delivered over bus 2144 to stage 2, and the memory write hardware forms the memory address. The memory address is then delivered over bus 2152 to the third stage which reads memory over 2164 through network 650 to the data cache interconnection network 1590. The results of the read operation are then stored and delivered to stage 4 for storage in register R0. Stage 4 delivers the data over lines 2172 through network 670 to register R0 in the register file. The same operation takes place for instruction I1 except that the results are stored in register 1. Hence, the four stages of the pipeline (Fetch, Form Memory Address, Perform Memory Read, and Store The Results) flow data through the pipe in the manner discussed, and when instruction I0 has passed through stage 1, the first stage

5,021,945

45

of instruction I1 commences. This overlapping or pipelining is conventional in the art.

Instruction I2 fetches the information stored in registers R0 and R1 in the register file 660 and delivers them into registers REG0 and REG1 of stage 1. The contents are delivered over bus 2144 through stage 2 as a no operation and then over bus 2152 into stage 3. A multiply occurs with the contents of the two registers, the results are delivered over bus 2162 into stage 4 which then stores the results over lines 2172 through network 670 into register R2 of the register file 660. In addition, the condition code data is stored over lines 2176 in the condition code storage of context files 660.

Instruction I3 performs the addition of the data in registers R2 and R3 in the same fashion, to store the results, at stage 4, in register R3 and to update the condition code data for that instruction. Finally, instruction I4 operates in the same fashion except that stage 3 performs a decrement of the contents of register R4.

Hence, according to the example of Table I, the instructions for PE0, would be delivered from the PIQ0 in the following order: I0, I2, and I3. These instructions would be sent through the PE0 pipeline stages (S1, S2, S3, and S4), based the upon the instruction firing times (T16, T17, and T18), as follows:

TABLE 20

PE	Inst	T16	T17	T18	T19	T20	T21
PE0:	I0	S1	S2	S3	S4		
	I2		S1	S2	S3	S4	
	I3			S1	S2	S3	S4
PE1:	I1	S1	S2	S3	S4		
PE2:	I4	S1	S2	S3	S4		

The schedule illustrated in Table 20 is not however possible unless data chaining is introduced within the pipeline processor (intraprocessor data chaining) as well as between pipeline processors (interprocessor data chaining). The requirement for data chaining occurs because an instruction no longer completely executes within a single time cycle illustrated by, for example, instruction firing time T16. Thus, for a pipeline processor, the TOLL software must recognize that the results of the store which occurs at stage 4 (T19) of instructions I0 and I1 are needed to perform the multiply at stage 3 (T19) of instruction I2, and that fetching of those operands normally takes place at stage 1 (T17) of instruction I2. Accordingly, in the normal operation of the pipeline, for processors PE0 and PE1, the operand data from registers R0 and R1 is not available until the end of firing time T18 while it is needed by stage 1 of instruction I2 at time T17.

To operate according to the schedule illustrated in Table 20, additional data (chaining) paths must be made available to the processors, paths which exist both internal to the processors and between processors. These paths, well known to those practiced in the art, are the data chaining paths. They are represented, in FIG. 21, as dashed lines 2180 and 2182. Accordingly, therefore, the resolution of data dependencies between instructions and all scheduling of processor resources which are performed by the TOLL software prior to program execution, take into account the availability of data chaining when needed to make available data directly from the output, for example, of one stage of the same processor or a stage of a different processor. This data chaining capability is well known to those practiced in the art and can be implemented easily in the TOLL

46

software analysis by recognizing each stage of the pipeline processor as being, in effect, a separate processor having resource requirements and certain dependencies, that is, that an instruction when started through a pipeline will preferably continue in that same pipeline through all of its processing stages. With this in mind, the speed up in processing can be observed in Table 20 where the three machine cycle times for the basic block are completed in a time of only six pipeline cycles. It should be borne in mind that the cycle time for a pipeline is approximately one-fourth the cycle time for the non-pipeline processor in the illustrated embodiment of the invention.

The pipeline of FIG. 21 is composed of four equal (temporal) length stages. The first stage 2140 performs the instruction decode, determines what registers to fetch and store, and performs up to two source register fetches which can be required for the execution of the instruction.

The second stage 2150 is used by the computational instructions for the condition code fetch if required. It is also the effective address generation stage for the memory interface instructions.

The effective address operations that are supported in the preferred embodiment of the invention are:

1. Absolute address
    The full memory address is contained in the instruction.
2. Register indirect
    The full memory address is contained in a register.
3. Register indexed/based
    The full memory address is formed by combining the designated registers and immediate data.
    a. Rn op K
    b. Rn op Rm
    c. Rn op K op Rm
    d. Rn op Rm op K

where "op" can be addition (+), subtraction (−), or multiplication (*) and "K" is a constant.

As an example, the addressing constructs presented in the matrix multiply inner loop example are formed from case 3-a where the constant "K" is the length of a data element within the array and the operation is addition (+).

At a conceptual level, the effective addressing portion of a memory access instruction is composed of three basic functions; the designation and procurement of the registers and immediate data needed for the calculation, the combination of these operands in order to form the desired address, and if necessary, updating of any one of the registers involved. This functionality is common in the prior art and is illustrated by the autoincrement and autodecrement modes of addressing available in the DEC processor architecture. See, for example, DEC VAX Architecture Handbook.

Aside from the obvious hardware support required, the effective addressing is supported by the TOLL software, and impacts the TOLL software by adding functionality to the memory accessing instructions. In other words, an effective address memory access can be interpreted as a concatenation of two operations, the first being the effective address calculation and the second being the actual memory access. This functionality can be easily encoded into the TOLL software by

BIA0001197

5,021,945

47

one skilled in the art in much the same manner as an add, subtract or multiply instruction would be.

The described effective addressing constructs are to be interpreted as but one possible embodiment of a memory accessing system. There are a plethora of other methods and modes for generating a memory address that are known to those skilled in the art. In other words, the effective addressing constructs described above are for design completeness only, and are not to be construed as a key element in the design of the system.

Referring to FIG. 22, various structures of data or data fields within the pipeline processor element of FIG. 21 are illustrated for a system which is a multi-user system in both time and space. As a result, across the multiple pipelines, instructions from different users may be executing, each with its own processor state. As the processor state is not typically associated with the processor element, the instruction must carry along the identifiers that specify this state. This processor state is supported by the LRD, register file and condition code file assigned to the user.

A sufficient amount of information must be associated with each instruction so that each memory access, condition code access or register access can uniquely identify the target of the access. In the case of the registers and condition codes, this additional information constitutes the absolute value of the procedural level (PL) and context identifiers (CI) and is attached to the instruction by the SCSM attachment unit 1650. This is illustrated in FIGS. 22a, 22b and 22c respectively. The context identifier portion is used to determine which register or condition code plane (FIG. 6) is being accessed. The procedural level is used to determine which procedural level of registers (FIG. 13) is to be accessed.

Memory accesses also require that the LRD that supports the current user be identified so that the appropriate data cache can be accessed. This is accomplished through the context identifier. The data cache access further requires that a process identifier (PID) for the current user be available to verify that the data present in the cache is indeed the data desired. Thus, an address issued to the data cache takes the form of FIG. 22d. The miscellaneous field is composed of additional information describing the access, for example, read or write, user or system, etc.

Finally, due to the fact that there can be several users executing across the pipelines during a single time interval, information that controls the execution of the instructions, and which would normally be stored within the pipeline, must be associated with each instruction instead. This information is reflected in the ISW field of an instruction word as illustrated in FIG. 22a. The information in this field is composed of control fields like error masks, floating point format descriptors, rounding mode descriptors, etc. Each instruction would have this field attached, but, obviously, may not require all the information. This information is used by the ALU stage 2160 of the processor element.

This instruction information relating to the ISW field, as well as the procedural level, context identification and process identifier, are attached dynamically by the SCSM attacher (1650) as the instruction is issued from the instruction cache.

Although the system of the present invention has been specifically set forth in the above disclosure, it is to be understood that modifications and variations can be

48

made thereto which would still fall within the scope and coverage of the following claims.

We claim:

1. A machine implemented method for parallel processing in a plurality of processor elements natural concurrencies in streams of low level instructions contained in programs of a plurality of users, each of the streams having a plurality of single entry-single exit (SESE) basic blocks (BBs), said method comprising the steps of:

statically adding intelligence representing the natural concurrencies existing within the instructions in each basic block of the programs, said step of adding for each program comprising the steps of:

(a) ascertaining resource requirements of each instruction within each basic block to determine the natural concurrencies in each basic block,

(b) identifying logical resource dependencies between instructions,

(c) assigning condition code storage (CCs) to groups of resource dependent instructions, such that dependent instructions can execute on the same or different processor elements,

(d) determining the earliest possible instruction firing time (IFT) for each of said instruction in each of said plurality of basic blocks,

(e) adding said instruction firing times to each instruction in each of said plurality of basic blocks,

(f) assigning a logical processor number (LPN) to each instruction in each of said basic blocks,

(g) adding said logical processor numbers to each instruction in each of said basic blocks, and

(h) repeating steps (a) through (g) until all basic blocks are processed for each of said programs, and

processing the instructions having the statically added intelligence for executing said programs on a plurality of processor elements (PEs).

2. The method of claim 1 wherein said step of statically adding intelligence further comprises the step of re-ordering said instructions in each of said basic blocks based upon said instruction firing times wherein the instructions having the earliest firing times are listed first.

3. A machine implemented method for parallel processing, in a system, natural concurrencies in streams of low level instructions contained in a program, each of the streams having a plurality of single entry-single exit (SESE) basic blocks (BBs), said system having a plurality of processor elements, said method comprising the steps of:

statically adding intelligence representing the natural concurrencies existing within the instructions in each basic block, said step of adding comprising the steps of:

(a) ascertaining resource requirements of each instruction within each basic block to determine the natural concurrencies in each basic block,

(b) identifying logical resource dependencies between instructions,

(c) assigning condition code storage (CCs) to groups of resource dependent instructions, such that dependent instructions can execute on the same or different processor elements,

(d) determining the earliest possible instruction firing time (IFT) for each of said instructions in each of said plurality of basic blocks,

(e) adding said instruction firing time (IFT) to each instruction in each of said plurality of basic blocks,

BIA0001198

5,021,945

49   **50**

(f) assigning a logical processor number (LPN) to each instruction in each of said basic blocks,

(g) adding said logical processor numbers to each instruction in each of said basic blocks, and

(h) repeating steps (a) through (g) until all basic blocks are processed for said program, and

processing the instructions having the statically added intelligence using a plurality of processor elements (PEs).

4. The method according to claim 3 wherein said step of statically adding intelligence further comprises the step of reordering said instructions in each of said basic blocks based upon said instruction firing times wherein the instructions having the earliest firing times are listed first.

5. The method according to claim 3 wherein said step of statically adding intelligence further comprises the step of adding program level information to identify the program level of said instruction.

6. A machine implemented method for parallel processing natural concurrencies in a program using a plurality of processor elements (PEs), said program having a plurality of single entry-single exit (SESE) basic blocks (BBs) with each of said basic blocks (BBs) having a stream of instructions, said method comprising the steps of:

determining the natural concurrencies within said instruction stream in each of said basic blocks (BBs) in said program,

adding intelligence to each instruction in each said basic block in response to the determination of said natural concurrencies, said added intelligence at least comprising an instruction firing time (IFT) and a logical processor number (LPN) so that all processing resources required by any given instruction are allocated in advance of processing, and

processing the instructions having said added intelligence in said plurality of processor elements corresponding to the logical processor numbers, each of said plurality of processor elements receiving all instructions for that processor in the order of the instruction having earliest instruction firing times, the instruction having the earliest time being delivered first.

7. The method of claim 6 wherein each instruction has a static shared context storage mapping information, and wherein the step of adding intelligence further comprises the step of adding static shared context storage mapping (S-SCSM) information and wherein the step of processing further comprises the step of processing each instruction requiring the shared resources as identified by each said instruction's static shared context storage mapping information, so each program routine can access resources at other procedural levels in addition to the resources at that routine's procedural level.

8. The method of claim 6 wherein the step of determining the natural concurrencies within each basic block comprises the steps of:

ascertaining the resource requirements of each instruction within each of said basic blocks,

identifying logical resource dependencies between instructions, and

assigning condition code storage (CCs) to groups of resource dependent instructions.

9. The method of claim 6 wherein the step of adding intelligence to each instruction further comprises the steps of:

determining the earliest possible instruction firing time for each of said instructions in each of said plurality of basic blocks,

adding said instruction firing times (IFTs) to each instruction in each of said plurality of basic blocks in response to said determination, and

reordering said instructions in each of said basic blocks based upon said instruction firing times.

10. The method of claim 6 wherein the step of adding intelligence to each instruction further comprises the steps of:

determining the earliest possible instruction firing time for each of said instructions in each of said plurality of basic blocks, and

adding said instruction firing times (IFTs) to each instruction in each of said plurality of basic blocks in response to said determination.

11. The method according to claim 10 further comprising the steps of:

assigning a logical processor number to each instruction in each of said basic blocks, and

adding said assigned logical processor number to each instruction in each of said basic blocks in response to said assignment.

12. The method of claim 6 further comprising the step of forming execution sets (ESs) of basic blocks in response to said step of adding said intelligence wherein branches from any given basic block within a given execution set to a basic block in another execution set is statistically minimized.

13. The method of claim 6 wherein the step of processing further comprises the steps of:

separately storing the instructions with said added intelligence based on the logical processor number, each group of said separately stored instructions containing instructions having only the same logical processor number,

selectively connecting said separately stored instructions to said processor elements, and

each said processor element receiving each instruction assigned to it with the earliest instruction firing time first.

14. The method of claim 13 wherein the step of processing said instruction received by an individual processor element further comprises the steps of:

obtaining the input data for processing said received instruction from shared storage locations identified by said instruction,

storing the results from processing said received instruction in a shared storage identified by said instruction, and

repeating the aforesaid steps for the next instruction until all instructions are processed.

15. A machine implemented method for parallel processing, in a system, natural concurrencies in a plurality of programs of different users with a plurality of processor elements (PEs), each of said programs having a plurality of single entry-single exit (SESE) basic blocks (BBs), with each of said basic blocks (BBs) having a stream of instructions, said method comprising the steps of:

determining the natural concurrencies within said instruction stream in each of said basic blocks in each said program,

adding intelligence to each instruction in each said basic block in response to the determination of said natural concurrencies, said added intelligence representing at least an instruction firing time (IFT)

5,021,945

51

and a logical processor number (LPN) so that all processing resources required by any given instruction are allocated in advance of processing,

processing the instructions having said added intelligence from said programs in said plurality of processor elements corresponding to the logical processor numbers, each of said plurality of processor elements receiving instructions in the order starting with the instruction having earliest instruction firing time, each said processor element being capable of processing instructions from said programs in a predetermined order.

16. The method of claim 15 in which the step of processing further comprises the steps of: dynamically adding context information to each instruction, said dynamically added information identifying the context file in said system assigned to each said program, each of said processor elements being further capable of processing each of its instructions using only the context file identified by said added context information.

17. The method of claim 15 wherein the step of adding intelligence further comprises the adding of program level information to each instruction involved in program level transfers and wherein the step for processing further comprises processing each instruction in communication with a set of shared registers was identified by said instruction's program level information, so that a program routing can access other sets of shared registers in addition to the routine's procedural level set of registers.

18. The method of claim 15 wherein the step of determining the natural concurrencies within each basic block of each program comprises the steps of:

ascertaining the resource requirements of each instruction within each of said basic blocks,

identifying logical resource dependencies between instructions, and

assigning condition code storage (CCs) to groups of resource dependent instructions, so that dependent instructions can execute on the same or different processor elements.

19. The method of claim 15 wherein the step of adding intelligence to each instruction in each said program further comprises the steps of:

determining the earliest possible instruction firing time for each of said instructions in each of said plurality of basic blocks,

adding said instruction firing times to each instruction in each of said plurality of basic blocks in response to said determination, and

reordering said instructions in each of said basic blocks based upon said instruction firing times.

20. The method of claim 15 wherein the step of adding intelligence to each instruction in each said program further comprises the steps of:

determining the earliest possible instructions firing time for each of said instructions in each of said plurality of basic blocks, and

adding said instruction firing times to each instruction in each of said plurality of basic blocks in response to said determination.

21. The method according to claims 19 or 20 further comprising the steps of:

assigning a logical processor number to each instruction in each of said basic blocks, and

adding said assigned logical processor number to each instruction in each of said basic blocks in response to said assignment.

52

22. The method of claim 15 further comprising the step of forming execution sets (ESs) of the basic blocks for each said program in response to said step of adding said intelligence wherein branches from any given basic block within a given execution set to a basic block in another execution set is statistically minimized.

23. The method of claim 15 wherein the step of processing further comprises the steps of:

separately storing the instructions with said added intelligence in a plurality of sets and wherein said separate storage of each set is based on the assigned logical processor number,

selectively connecting said separately stored instructions in dais sets to said processor elements based on said logical processor number, and

each said processor element receiving the instruction having the earliest instruction firing time first.

24. The method of claim 23 wherein the step of receiving said instruction from said separately stored instructions in said sets by an individual processor element further comprises the steps of:

obtaining input data for processing said received instruction from shared storage locations identified by said instruction,

storing the results based from processing said received instruction in a shared storage identified by said received instruction, and

repeating the aforesaid steps for the next received instruction from said next set based upon said predetermined order until all instructions are processed in said sets.

25. A method for parallel processing natural concurrencies in a plurality of programs with a plurality of processor elements (PEs), said processor elements having access to input data located in a plurality of shared resource locations, each of said programs having a plurality of single entry-single exit (SESE) basic blocks (BBs) with each of said basic block s (BBs) having a stream of instructions, said method comprising the steps of:

ascertaining the resource requirements of each instruction within each of said basic blocks for each said program,

determining, based on said resource requirements, the earliest possible instruction firing time (IFT) for each of the instructions in each of said plurality of basic blocks,

adding said instruction firing times to each instruction in each of said plurality of basic blocks in response to said determination,

assigning a logical processor number (LPN) to each instruction in each of said basic blocks,

adding said assigned logical processor number to each instruction in each of said plurality of basic blocks in response to said assignment,

separately storing the instructions with said added instruction firing time and said added logical processor numbers in a plurality of sets, each set having a plurality of separate storage regions, wherein at least one set contains at least one of said programs and wherein said separate storage in each set is based on the logical processor number,

selectively connecting said separately stored instructions for one of said sets to said processor elements based on the logical processor number, and

each said processor element receiving each instruction to be connected thereto with the earliest in-

53

struction firing time (FT) first, said processor element being capable of performing the steps of:

(a) obtaining said input data for processing said received instruction from a shared storage location in said plurality of shared resource locations identified by said instruction,

(b) storing the results based from processing said received instruction in a shared storage location in said plurality of shared resource locations identified by said received instruction, and

(c) repeating the aforesaid steps (a) and (b) for the next received instruction from one of said sets based upon a predetermined order until all instructions are processed.

26. The method of claim 25 further comprising the steps of forming execution sets (ESs) of basic blocks in response to said steps of adding said instruction firing times and logical processor numbers wherein branches from any given basic block within a given execution set to a basic block in another execution set is statistically minimized.

27. The method of claim 25 wherein the step of adding intelligence further comprises the step of adding static shared context storage mapping information and wherein the step of processing further comprises the step of processing each instruction requiring at least one shared storage location, said at least one location corresponding to said instruction's static shared context state mapping information, so each program routine can access its set of storage locations and, in addition, at least one set of shared storage locations at another procedural level.

28. A system for parallel processing natural concurrencies in a program, said program having a plurality of single entry-single exit (SESE) basic blocks (BBs) wherein each of said basic blocks (BBs) contains a stream of instructions, said system comprising:

means receptive of said plurality of basic blocks for determining said natural concurrencies within said instruction stream for each of said basic blocks, said determining means further adding at least an instruction firing time (IFT) and a logical processor number (LPN) to each instruction in response to said determined natural concurrencies so that all processing resources required by any given instruction are allocated in advance of processing,

means receptive of said basic blocks having said added instruction firing times and logical processor numbers for separately storing said received instructions based upon said logical processor number, and

a plurality of processor elements (PEs), connected to said storing means based upon said logical processor numbers, each of said processor elements processing its received instructions and the order of processing those instructions being that the instructions with the earliest instruction firing time are processed first.

29. The parallel processor system of claim 28 in which:

said determining means further has means for adding program level information (S-SCSM) to said instructions, said information containing level information for each said instruction in order to identify the relative program levels required by the instruction within said program,

54

a plurality of sets of registers, said registers having a different set of registers associated with each said program level, and

said processor elements being further capable of processing each of its received instructions in at least one set of registers identified by said received instruction.

30. The system of claim 28 wherein said determining means further has means for forming the basic blocks containing said added instruction firing times and logical processor numbers into execution sets (ESs), wherein branches from any given basic block within a given execution set out of said given execution set to a basic block in another execution set is statistically minimized.

31. The system of claim 30 wherein said determining means is further capable of attaching header information to each formed execution set, said header at least comprising:

(a) the address of the start of said instructions, and

(b) the length of the execution set.

32. The system of claim 30 further comprising storing means having:

a plurality of caches receptive of said execution sets for storing said instructions,

means connected to said caches for delivering said instructions stored in each of said caches to said plurality of processor elements (PEs), and

means connected to said caches and said delivering means for controlling said storing and said delivery of instructions, said controlling means being further capable of executing the branches from individual basic blocks.

33. The system of claim 28 wherein each of said plurality of processor elements is context free in processing a given instruction.

34. The system of claim 28 wherein said plurality of shared resources comprises:

a plurality of register files, and

a plurality of condition code files, said plurality of register files and said plurality of condition code files together with said storing means being capable of storing all necessary context information for any given instruction during the processing of said instruction.

35. A machine implemented method for parallel processing streams of low level instructions contained in the programs of a plurality of users for execution in a system having a plurality of processor elements and shared storage locations, each of the streams having a plurality of single entry-single exit basic blocks, said method comprising the steps of:

statically adding intelligence to the instructions in each basic block of the programs, said added intelligence being responsive to natural concurrencies within said instruction streams, said added intelligence representing at least an instruction firing time for each said instruction, and

processing the instructions having the statically added intelligence for executing the programs, the step of processing further comprising the steps of:

(a) delivering the instructions to the system,

(b) assigning each said user program to a context file in said system, whereby each program during execution has its own identifiable context file,

(c) dynamically generating shared context storage mapping information for said instructions identify-

5,021,945

55

56

ing said context file being assigned to said instructions and containing shared storage locations,

(d) separately storing the delivered instructions in the system based on a processor on which the instruction is to be executed,

(e) selectively connecting the separately stored instructions to the assigned processor for the instructions, said separately stored instructions being delivered in order of earliest instruction firing times of the separately stored instructions,

(f) sequentially processing said instructions from each connected separately stored instructions in each said connected processor element,

(g) obtaining any input data for processing said connected instruction from a shared storage location identified, at least in part, by said shared context storage mapping information,

(h) storing the results of processing a said connected instruction in a shared storage location identified at least in part by said shared context storage mapping information, and

(i) repeating steps (a) through (h) until all instructions of each of aid plurality of basic blocks for all of said programs are processed.

36. A machine implemented method for parallel processing a stream of instructions having natural concurrencies in a parallel processing system having a plurality of processor elements, said stream of instructions having a plurality of single entry-single exit basic blocks, said method comprising the steps of:

determining natural concurrencies within said instruction stream,

adding intelligence to each instruction stream in response to the determination of said natural concurrencies, said added intelligence comprising at least an instruction firing time and a logical processor

number for each instruction so that all processing resources required by any given instruction are allocated in advance of processing, and

processing the instructions having said added intelligence in said plurality of processor elements corresponding to the logical processor number, each of said plurality of processor elements receiving all instructions for that processor element in order according to the instruction firing times, the instruction having the earliest time being delivered first.

37. A machine implemented method for parallel processing a plurality of programs of different users in a system having a plurality of processor elements, each of said programs having a plurality of single entry-single exit basic blocks, with each of said basic blocks having a stream of instructions, said method comprising the steps of:

determining the natural concurrencies among the instructions in each said program,

adding intelligence to said basic blocks in response to the determination of said natural concurrencies, said added intelligence comprising at least an instruction firing time and a logical processor number so that all processing resources required by any given instruction are allocated in advance of processing, and

processing the instructions having said added intelligence in said plurality of processor elements corresponding to the logical processor number, each of said plurality of processor elements receiving instructions in accordance with the instruction firing times, starting with the instruction having earliest instruction firing time.

* * * * *

40

45

50

55

60

65

BIA0001202